# A Tool-Assisted Approach to Engineer Domain-Specific Languages (DSLs) using Rust

Léo Olivier
Nantes Université
Nantes, France
leo.olivier@etu.univ-nantes.fr

Lou-Anne Sauvêtre
Nantes Université
Nantes, France
lou-anne.sauvetre@etu.univ-nantes.fr

Erwan Bousse
LS2N, Nantes Université
Nantes, France
erwan.bousse@ls2n.fr

Gerson Sunyé
LS2N, Nantes Université
Nantes, France
gerson.sunye@ls2n.fr

## ABSTRACT

Domain-Specific Languages (DSLs) are required in a wide range of contexts, implying different execution environments. The same DSL may even have to exist in different environments. We propose in this paper an approach to engineer a DSL using the Rust language, which can be used to target several environments. Our approach focuses on selecting an implementation language, Rust, that provides multiple compilation targets for a DSL definition. However, this is a rather laborious process because Rust is only partially object-oriented, while the definition of a metamodel-based abstract syntax is essentially object-oriented. For this reason, we offer a complete DSL's development method beginning with the metamodel definition in Ecore language, then the abstract syntax conversion in Rust with a code generation tool, and finally the deployment of the language in different execution environments. We evaluated our approach by creating two DSLs with it, a Petri nets DSL and a Finite State Machine (FSM) DSL. Finally, we discuss possible improvements for our Ecore2Rust conversion tool.

## KEYWORDS

Model-Driven Engineering, Software Language Engineering, Domain-Specific Languages, Eclipse Modeling Framework, Rust

**ACM Reference Format:**
Léo Olivier, Lou-Anne Sauvêtre, Erwan Bousse, and Gerson Sunyé. 2022. A Tool-Assisted Approach to Engineer Domain-Specific Languages (DSLs) using Rust. In *ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MODELS '22 Companion), October 23–28, 2022, Montreal, QC, Canada.* MIL Campus of the University of Montreal, Montréal, Canada, 10 pages. https://doi.org/10.1145/3550356.3563133

## 1 INTRODUCTION

*"If a particular kind of problem occurs often enough, then it might be worthwhile to express instances of the problem as sentences in a simple language"* [5]. Defining a Domain-Specific Language (DSL) [4, 10, 13] is a technique widely used in the software industry to help developping readable, maintainable and flexible applications. Well-known DSLs include SQL and HTML, but typical simpler examples include Petri nets or Finite State Machines (FSM).

There are nowadays manifold programming languages, frameworks and language workbenches to engineer new DSLs [3], such as Rascal [8], MetaEdit+ [7], the GEMOC Studio [2] or MPS [1], to name a few. In the realm of modeling languages, a most popular DSL development environment is the Eclipse Modeling Framework (EMF) [12], which allows to create efficiently metamodel-based abstract syntaxes using the Ecore language. EMF targets Java as an execution environment: its code generation tool can only convert Ecore models to Java source code. However, software industry needs are evolving and the deployment of the same DSL may occur in multiple execution environments such as embedded systems and web applications. Unfortunately, to our knowledge, there is currently no development environment that meets those needs.

In this context, an interesting idea would be to implement a DSL using the Rust programming language, which has the following features: memory safety, safe parallelism, efficient memory management, and no garbage collector. Those attributes produce 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.

However, Rust comes with a handful of a rough edges. Thus, defining a consistent and error-free abstract syntax in Rust is a difficult task, especially when aiming for a metamodel-based one. While specific design patterns exist to overcome these difficulties, they remain laborious to implement.

To overcome these problems, we propose a complete approach for creating and deploying a DSL, starting with the definition of an abstract syntax in the form of an Ecore metamodel by the language engineer. Then, the abstract syntax is converted into Rust code thanks to our tool called Ecore2Rust. Then, the operational semantics must be manually programmed in Rust by the language engineer. At the end of this process, a Rust DSL is produced and can be deployed in various execution environments, as a CLI tool


*MODELS '22 Companion, October 23–28, 2022, Montreal, QC, Canada*

ACM ISBN 978-1-4503-9467-3/22/10…$15.00
https://doi.org/10.1145/3550356.3563133

[1] http://jetbrains.com/mps

on desktop or a web application through the binary format WebAssembly. A domain expert can thereby use the DSL in different execution environments.

To evaluate our contribution, we applied our approach to engineering complete Rust-based Petri nets and FSM DSLs. We show that both DSLs can be deployed as a web application and CLI tool, making it available from any connected terminal through a web browser but also usable by offline users in the command line.

The rest of this paper is structured as follows. Section 2 introduces terms and concepts which are used in the paper. Furthermore, we detail why Rust is an interesting candidate for implementing DSLs. Section 3 gives an overview of our contribution and technical choices. Section 4 provides a description of the experimental protocol and discusses our result. Section 5 discusses the related work. The conclusion in Section 6 sums up our project and proposes various improvement axes for our work.

## 2 BACKGROUND & MOTIVATION

In this section, we define what is a metamodel-based DSL, its usage, and we motivate our approach using an illustrative example. Then, we explain the opportunities offered by Rust to deploy DSLs in various execution environments.

### 2.1 DSLs Considered in This Work

A DSL is a language providing abstractions for a particular domain. It is used as a high-level tool to increase software development productivity. In what follows, we present the main components of a DSL considered in this paper.

*Abstract syntax and editing operations.* The central piece of a DSL is the abstract syntax of the language, which formally defines the rules to create a syntactically valid model. One way to define an abstract syntax is to create a metamodel using a metamodeling language such as using Ecore from the Eclipse Modeling Framework (EMF) [12]. To supplement the abstract syntax, a DSL can rely on a set of editing operators. These editing operators can be used to create and connect instances of the concepts of the abstract syntax, and are typically used to define a proper model editor for the DSL.

Figure 1 shows an example of abstract syntax of a Petri nets DSL. A Petri net model consists of *Places*, *Transitions* and directed *Edges* which are part of a network. *Places* can contain tokens. *Edges* have a weight and can only connect a *Place* and a *Transition* to each other. The *Orientation* enumeration indicates if an *Edge* is directed towards a *Place* or a *Transition*. *Places* and *Transitions* have a list of input and output *Edges*.

Figure 2 shows an example of a Petri nets model using a graphical concrete syntax. Places are represented by circles, transitions by rectangles, and edges by arrows. The dots in the circles indicate the number of tokens in the places.

Editing operations for this Petri Nets DSL would typically include an operation to create a new *Place*, an operation to create a *Transition*, and an operation to connect a *Place* and a *Transition* through an edge.

*Execution semantics.* The execution semantics of a DSL can be defined in at least two different ways: as an operational semantics (i. e. an interpreter) or as a translational semantics (i. e. a compiler).

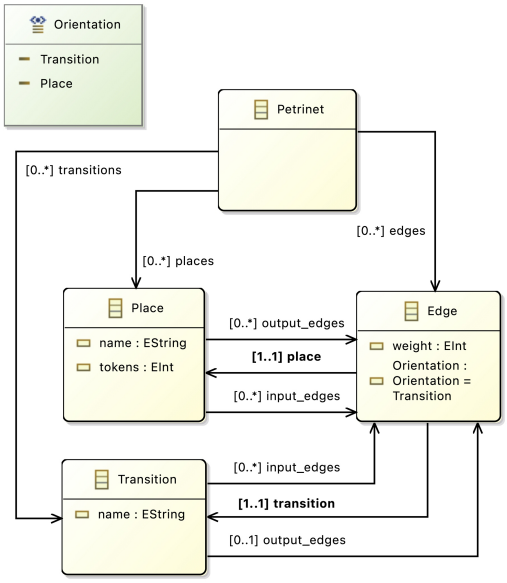


**Figure 1: Petri nets Abstract Syntax in Ecore**

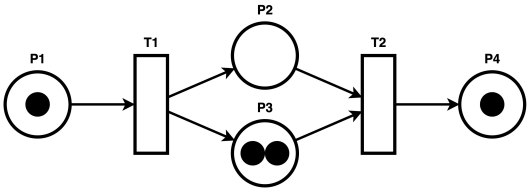


**Figure 2: Petri nets Model Example**

**Algorithm 1:** The Method Firing a Transition

**input :** A Petri nets $P$ and a transition $trans$ with $trans \in P$
**output :** A boolean: $true$ if the transition has been fired, $false$ otherwise

```
1  begin
2      if isFirePossible(trans) is false then
3          return false
4      foreach edge ∈ trans.inputEdges do
5          place ← edge.place
6          place.tokens ← place.tokens − edge.weight
7      foreach edge ∈ trans.outputEdge do
8          place ← edge.place
9          place.tokens ← place.tokens + edge.weight
10     return false
```

In this work we focus on operational semantics, even though the approach could be easily adapted to translational semantics.

Algorithms 1 and 2 describe the operational semantics for the Petri nets DSL introduced just previously. Algorithm 1 describes the firing of a transition: it consumes the required input tokens (defined by the *Edge* weight), and creates tokens in its output *Places*.

**Algorithm 2:** The method checking if it is possible to fire a transition

```
input  : A Petri nets P and a transition trans with trans ∈ P
output : A boolean: true if the transition can be fired, false
         otherwise
1 begin
2     foreach edge ∈ trans.inputEdges do
3         if edge.weight > edge.place.tokens then
4             return false
5     return true
```

The possibility of firing the transition is checked beforehand as described by the Algorithm 2.

## 2.2 Rust and its Usage for DSL Implementation

Our work focuses on the opportunities offered by Rust to create and deploy such DSLs in different sorts of execution environments, to provide end-user tools (editor, executor, etc.) for these different environments.

*2.2.1 Main Concepts and Ideas of Rust.* Rust is a fairly-new compiled programming language designed for performance, memory safety, and thread safety. It is a strong and static typed language. It is said to be multi-paradigms: it carries concepts from functional, imperative, and object-oriented programming[2]. It was announced by the Mozilla Foundation in 2010 and its development is now under the support of the Rust Foundation. It has a growing community and is, according to the yearly StackOverflow survey, the most loved language since 2016[3].

Rust allows low-level memory management and safe parallelism by a set of rules checked by the compiler based on the concept of ownership. That way, Rust does not rely on a garbage collector. The ownership system can be resumed as follows: each value is owned by at most one variable at a time and when the owner goes out of scope, the value is dropped. Those attractive properties are useful in DSL development: strong typing, memory safety and multiple deployment targets.

*2.2.2 Compilation Targets.* As Rust is oriented towards network services, embedded systems, command line tools, and web applications, it has multiple compilations targets and can run on almost all platforms[4].

Rust provides helpful features to create a command-line application, such as automatically-generated documentation[5], quick packaging and distributing thanks to the Cargo package manager[6], and flexible logging configuration[7].

For the development of applications running on embedded systems, Rust emphasizes interoperability and portability: it can be integrated into an existing C codebase[8] and it provides an efficient Hardware Abstraction layer, allowing to use a driver or a library with a wide variety of systems[9].

A key benefit of implementing a DSL using Rust is that Rust code can be compiled into a web-runnable binary instruction format called WebAssembly (abbreviated Wasm). It allows the execution of applications written in these languages on the web nearing native performances, taking advantage of hardware capabilities available on computer, mobile, and IoT devices.

WebAssembly is designed to be memory safe, portable, fast, and fully integrated. It does not replace JavaScript, which remains the language for the web, but exposes its modules and offers an API for communicating with JavaScript. It is possible to build an entire application with WebAssembly, user interfaces included, or to only make a library of utility functions called by JavaScript. Although web deployment is WebAssembly's key focus, it can be shipped on non-web embeddings such as Node and used on the server side for edge computing, for example.

For example, in the context of DSL development, a language engineer could compile a DSL developed using Rust that his company uses into a Wasm binary and can make it run on the web without rewriting the language for each execution environment.

## 2.3 Shortcomings of Rust for Metamodeling

Despite Rust interesting features, some complications arise when implementing DSLs in this fashion. This is especially apparent when implementing a metamodel-based abstract syntax: because of the way Rust manages memory, it is tricky to replicate bidirectional associations and shared references. Indeed, the Rust low-level memory management enforces thinking in terms of heap and stack allocation, whereas metamodeling is a high-level activity that should allow abstract concepts to be expressed simply. By default, all values in Rust are allocated on the stack because managing memory is more efficient. On the contrary, heap allocation is used for dynamically sized data structures, such as vector arrays, or when programmers explicitly need to control the variable lifetime.

In addition, while Rust does share some similarities with object-oriented programming, it is not built around a class taxonomy principle. That makes converting an Ecore-built language into idiomatic Rust not straightforward. For instance, Rust has no concept of abstract classes, polymorphic methods, or class inheritance.

Solving those issues related to Rust design choices is laborious and requires a systematic approach. Furthermore, the Ecore language is more convenient for metamodeling as it is designed for it and simpler than Rust—it does not require thinking about memory management. For these reasons, we propose an approach to automate the conversion of a metamodel-based DSL expressed in Ecore into Rust code. This way, it is possible to take advantage of Ecore features while keeping the deployment opportunities offered by Rust. This approach is the subject of the following section.

# 3 APPROACH

In this section, we present the design of the approach and its envisioned use by language engineers. We motivate our design choices for the Ecore2Rust tool and detail how it fits into the approach we are proposing.

[2] https://doc.rust-lang.org/book/ch17-01-what-is-oo.html#characteristics-of-object-oriented-languages
[3] https://survey.stackoverflow.co/2022/#section-most-loved-dreaded-and-wanted-programming-scripting-and-markup-languages
[4] https://doc.rust-lang.org/nightly/rustc/platform-support.html
[5] https://rust-cli.github.io/book/in-depth/docs.html
[6] https://rust-cli.github.io/book/tutorial/packaging.html
[7] https://rust-cli.github.io/book/in-depth/human-communication.html
[8] https://docs.rust-embedded.org/book/interoperability/
[9] https://docs.rust-embedded.org/book/portability/

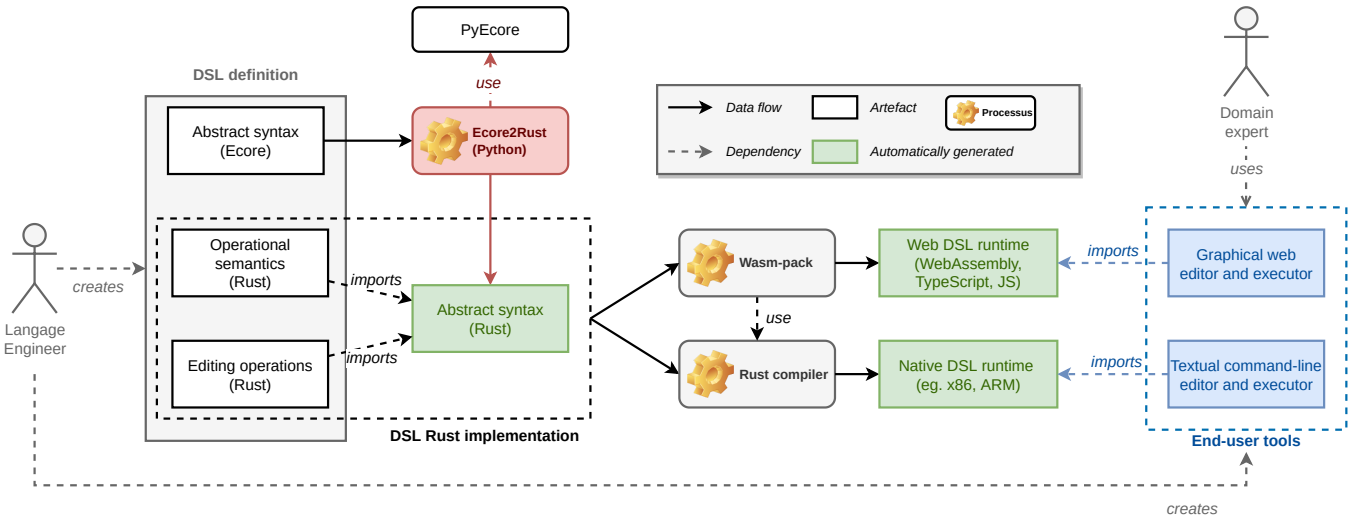


**Figure 3: Approach Overview**

## 3.1 Overview

Figure 3 depicts an overview of the proposed approach. From left to right, it shows the creation of the language and its usage in different execution environments.

First, the language engineer defines the abstract syntax of the DSL using the Ecore metamodeling language. Then, the Ecore2Rust tool translates this Ecore-based abstract syntax into pure Rust code, thus enabling the use of the abstract syntax in a Rust program. Finally, the language engineer defines both the operational semantics (i. e. the interpreter) and the editing operations of the DSL in Rust, using the generated Rust version of the abstract syntax.

Once a pure Rust DSL has been completed, the language engineer can rely on different Rust compilation toolchains to generate DSL runtimes for different execution environments. We call *DSL runtime* a software artifact that includes all the elements—libraries, executables, API, assets, etc.—that are necessary to integrate the DSL in a end-user tool in a given execution environment. For example, from the Rust implementation of the DSL, wasm-pack can be used to generate a DSL runtime fitting for web environments, and the Rust compiler can be used to generate a native DSL runtime.

Finally, using a given language runtime, the language engineer is able to create relevant end-user tools for the corresponding execution environment. For example, using the web runtime, a web-based graphical end-user tool can be made to edit and debug models, and using the native runtime, a textual command-line end-user tool can be made to execute and debug models in the command line. Both tools depend on the very same DSL definition, which guarantees that they remain consistent with one another.

## 3.2 Ecore2Rust

This subsection presents the central piece of our contribution, which is a tool to automatically generate Rust code from an abstract syntax defined using Ecore. The purpose of this tool is to reduce the amount of intricate Rust code that the language engineer should write to create a Rust-based DSL.

*3.2.1 Presentation.* Ecore2Rust performs the conversion of an Ecore model into corresponding Rust code. This conversion is not straightforward as Rust and Ecore do not rely on the same paradigms, goals, and constraints. Rust is partially object-oriented, while Ecore is fully object-oriented. Rust is a general-purpose programming language, while Ecore is mostly focused on defining data structures. Rust is considered to be rather "low-level" because it allows precise and safe memory management at the cost of more stringent allocation rules—i. e. the ownership system. On the contrary, Ecore is considered "high-level", and its official implementation relies on Java, and thus on a garbage collector. For these reasons, it was necessary to make technical choices regarding which Rust concepts to use to convert Ecore concepts.

Table 1 shows the mappings from Ecore to Rust considered for implementing Ecore2Rust. In what follows, we present and discuss most important aspects and challenges of this transformation.

*3.2.2 Objects in Rust.* Rust has the concept of objects owning data attributes and methods operating on these data. In idiomatic Rust, object attributes are grouped in *structs* and a method is an *implementation* of a function working on a *struct*. Shared behavior between *structs* is defined by *traits* that are similar to interfaces in other object-oriented languages. Thanks to *traits*, duck typing is allowed in Rust, which means it is possible to reproduce a kind of polymorphism. For example, if *structs* A and B implement *trait* C, then C can be used as a type accepting *structs* A and B.

*3.2.3 Classes.*

*Concrete Class.* Concrete classes of the Ecore abstract syntax are converted to Rust *structs*, and all their attributes and relationships to other classes become fields of the Rust *struct*.

*Asbtract Class.* As there is no abstract class and class inheritance in Rust, those classes are substituted by *traits*. A *trait* specifies abstract class methods and is implemented by the inherited concrete class. These are represented by *structs* and directly have inherited

| Ecore | Rust |
|---|---|
| EEnum | Enum |
| EClass | Struct |
| EClass (abstract) | Trait |
| Attribute | Detailed in **Table 3** |
| Reference | Rc<RefCell<T>> |
| Reference (containment) | T |
| Reference (recursive) | Box<T> |
| Reference (0 or 1) | Option<T> |
| Reference (abstract type) | Box<dyn T> |

**Table 1: Data Structure Conversion Summary Table**

| | Ownership | Invariants check | Borrows |
|---|---|---|---|
| Rc<T> | Multiple | Compile time | Immutable |
| Box<T> | Single | Compile time | Immutable or mutable |
| RefCell<T> | Single | Runtime | Immutable or mutable |

**Table 2: Some Smart Pointers Properties Explained**

attributes in them. Concrete class attributes containing an abstract type are allowed in Rust thanks to duck typing. But it needs to be prefixed by the dyn keyword because the compiler does not know the concrete type being passed. As it is a reference to a concrete type implementing a *trait*, it must be nested in a Box<dyn T> smart pointer[10].

*3.2.4 Reference.* A *reference* is a relation between two Ecore classes which means that a class A knows a class B. Each class exists independently of the other. Ecore2Rust converts all *references* to a regular Rust *struct* field.

An Ecore metamodel is essentially a graph data structure, which may include different sorts of single or bidirectional references. Unfortunately, Rust's ownership rules make building graph structures difficult. In particular, it is not straightforward to produce Rust data structures where there are bidirectional relations—i. e. shared ownership of data.

Although the Rust ownership system allows only one owner at a time for a value, there is a way to share a reference to this value with other variables. This is called *borrowing*[11] and it obeys the following rules: at any given time, there can be either one borrowed mutable reference, or any number of immutable borrowed references, and references must always be valid—unlike a pointer, a reference is guaranteed to point to a valid value of a particular type. However, there are some particular cases where special memory containers, called smart pointers, are needed[12].

To solve the graph structure issue, the state of the art in Rust is to use the *Interior Mutability Pattern*[13]. This design pattern consists in mutating the value inside an immutable value. It allows multiple owners of mutable data. In practice, it nests two kinds of smart pointers: RefCell<T> in Rc<T>. Note that T here corresponds to the type towards which we would like to have a pointer to.

[10] https://doc.rust-lang.org/book/ch19-04-advanced-types.html
[11] https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html
[12] https://doc.rust-lang.org/book/ch15-00-smart-pointers.html
[13] https://doc.rust-lang.org/book/ch15-05-interior-mutability.html

| Ecore | Rust |
|---|---|
| EByte | i8 |
| EShort | i16 |
| EInt | i32 |
| ELong | i64 |
| EDouble | f32 |
| EFloat | f64 |
| EBoolean | bool |
| EChar | char |
| EString | String |

**Table 3: Primitive Data Types Table**

Table 2 gives an overview of the properties of the three smart pointers we used in Ecore2Rust. We see that the combination of the Rc<T> and the RefCell<T> smart pointers makes it possible to have shared ownership thanks to the *Interior Mutability Pattern*.

The *Reference Counting* or Rc<T> is a smart pointer, keeping track of the number of references to a value. Cloning a Rc<T> increments the counter and creates a new reference to the value. When it goes to zero, the value is dropped. Rc<T> allows sharing data between multiple structures in the program, but is read-only.

Nesting a RefCell<T> in a Rc<T> makes the shared data mutable. RefCell<T> is another kind of smart pointer. It represents single ownership over the data it holds. It has the advantage of allowing immutable or mutable borrowings to be checked at runtime. Checking at runtime offers more possibilities for programs where compiler static analysis cannot ensure memory safety. Because of its properties, RefCell<T> lets us mutate the value inside, even when the RefCell<T> is immutable.

In summary, Ecore2Rust converts Ecore *references* following the *Interior Mutability Pattern*, producing Rust Rc<RefCell<T>> type.

*3.2.5 Containment Reference.* A *containment* is a particular case of *reference* defining that an instance of class A may contain an instance of class B. It is directly converted in a regular Rust field, hence directly typed with T.

*3.2.6 Zero-or-one Reference.* An Ecore metamodel can depict a situation where a class can have zero or one reference to an class. Instead of having a null type, Rust has an enumeration called *Option* which encodes the class of a value being present or absent[14]. Checking the existence of a value is done by pattern matching. In case of zero-or-one reference, the converter translates it into an Option<T> enum.

*3.2.7 Primitive Data Types.* Table 3 depicts the conversion table for primitive data types between Ecore and Rust. Some Ecore data types are not currently supported, such as EMap, and EObject variants. Ecore data types in Table 3 exactly match their Rust equivalent: for example, number types have the same ranges[15].

*3.2.8 Recursive Reference.* Recursive reference is when an Ecore class has a reference towards itself. Rust cannot know at compile

[14] https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html#the-option-enum-and-its-advantages-over-null-values
[15] https://doc.rust-lang.org/book/ch03-02-data-types.html

time how much space a recursive type takes, possibly looping infinitely. Nevertheless, recursive types are possible using a `Box<T>` smart pointer[16] because *Boxes* have a compile-time known size.

*3.2.9 Collection.* We call collection an Ecore reference with an upper bound superior to one. There are two kinds of array data structures in Rust: fixed-size arrays denoted `[T; N]` with T type and N size, and dynamic length arrays called Vector and denoted `Vec<T>`. The first one would be used in the case of a fixed-size collection greater than one. Otherwise, for an unknown-size collection, a `Vec<T>` would be used.

*3.2.10 Naming Convention.* Although it does not appear in the Table 1, it should be noted that Rust uses distinct name conventions for type-definition structures (types, *structs*, *traits*), and for variable names. Therefore, Ecore2Rust translates structure and variable names during conversion to match Rust conventions.

*3.2.11 Example of Ecore2Rust usage.* Figure 4 shows the generated Rust code for the abstract syntax of the Petri nets DSL depicted in Figure 1. Each Ecore class becomes a Rust *struct*, and all the class's attributes and relationships with the other classes become fields of the *struct*. *Structs* and their fields visibility are set to public with the keyword `pub`. *Structs* and *enums* names are written in pascal case (e. g. `Petrinet`, `Orientation`), while field names are written in snake case (e. g. `input_edges`), as required by Rust naming conventions. Lists of shared references such as `Petrinet`'s attributes, and `Place` and `Transition`'s `input_edges` and `output_edges` become a `Vec<Rc<RefCell<T>>>`. Simple shared references `place` and `transition` in `Edge` become a `Rc<RefCell<T>>`.

*3.2.12 Implementation of Ecore2Rust.* Ecore2Rust is implemented in Python and relies on the PyEcore framework[17] to read an Ecore metamodel previously serialized as an XMI file. It is freely accessible on a GitLab instance[18] under the AGPL license version 3.

In the current version of our prototype, we only support a subset of the Ecore language. Currently supported Ecore features are: enums, classes (abstract classes included), primitive data type attributes, subpackages, and references. Some parts of Ecore are not supported, such as methods and attributes inheritance, multiple inheritance, generic types, and Ecore's specific primitive data types. Although the import of used smart pointers is automatically added, the automatic import of *structs* between different subpackages is not supported.

## 3.3 Using the Generated Rust Abstract Syntax

Once the Rust code for the abstract syntax has been generated, it can be used by the language engineer as a basis to implement both editing operations and operational semantics in Rust.

Figure 5 shows the methods signatures of the editing operations manually written in Rust for the Petri nets DSL. These methods are in an *implementation* operating on the `Petrinet` *struct*. The `new()` method works as a constructor, returning a new `Petrinet` object. `add_transition()`, `add_place()` and `add_edge()` create respectively a new *Transition*, *Place*, and *Edge* in the Petri nets.

[16] https://doc.rust-lang.org/book/ch15-01-box.html
[17] https://github.com/pyecore/pyecore
[18] https://gitlab.univ-nantes.fr/E187954Y/ecore2rust

```rust
pub enum Orientation {
    Transition,
    Place,
}

pub struct Edge {
    pub place: Rc<RefCell<Place>>,
    pub transition: Rc<RefCell<Transition>>,
    pub orientation: Orientation,
    pub weight: i32,
}

pub struct Place {
    pub name: String,
    pub tokens: i32,
    pub input_edges: Vec<Rc<RefCell<Edge>>>,
    pub output_edges: Vec<Rc<RefCell<Edge>>>,
}

pub struct Transition {
    pub name: String,
    pub input_edges: Vec<Rc<RefCell<Edge>>>,
    pub output_edges: Vec<Rc<RefCell<Edge>>>,
}

pub struct Petrinet {
    pub places: Vec<Rc<RefCell<Place>>>,
    pub transitions: Vec<Rc<RefCell<Transition>>>,
    pub edges: Vec<Rc<RefCell<Edge>>>,
}
```

**Figure 4: Rust Abstract Syntax of the Petri Nets DSL, Generated from the Metamodel Shown in Figure 1.**

```rust
impl Petrinet {

  pub fn new() -> Petrinet {...}

  pub fn add_place(
    &mut self,
    name: String,
    tokens: i32
  ) -> Rc<RefCell<Place>> {...}

  pub fn add_transition(
    &mut self,
    name: String
  ) -> Rc<RefCell<Transition>> {...}

  pub fn add_edge(
    &mut self,
    rc_place: &Rc<RefCell<Place>>,
    rc_transition: &Rc<RefCell<Transition>>,
    orientation: Orientation,
    weight: i32,
  ) {...}

  ...

}
```

**Figure 5: Methods signatures of the Rust editing operations manually written for the Petri nets DSL.**

These methods have a special parameter named `self`, which is the Petri nets object itself on whose the methods are called.

Figure 6 shows the methods signatures of the operational semantics manually written in Rust for the Petri nets DSL. These methods are in the same *implementation* as the editing operations and have a similar `self` parameter. The method `is_fire_possible()` returns `true`, if it is possible to `fire` the transition given in parameter, and

```rust
impl Petrinet {

  ...

  pub fn is_fire_possible(
    &self,
    rc_transition: &Rc<RefCell<Transition>>
  ) -> bool {...}

  pub fn fire(
    &mut self,
    rc_transition: &Rc<RefCell<Transition>>
  ) -> String {...}

}
```

**Figure 6: Methods Signatures of the Rust Operational Semantics Manually Written for the Petri Nets DSL.**

```rust
pub fn fire(&mut self, rc_transition: &Rc<RefCell<Transition>>) -> bool {
    let is_fire_possible: bool = self.is_fire_possible(rc_transition);
    if !is_fire_possible {
        return false;
    }
    let trans = rc_transition.borrow();
    for edge in trans.input_edges.iter() {
        let place = &edge.borrow().place;
        place.borrow_mut().tokens -= edge.borrow().weight;
    }
    for edge in trans.output_edges.iter() {
        let place = &edge.borrow().place;
        place.borrow_mut().tokens += edge.borrow().weight;
    }
    return true;
}
```

**Figure 7: Implementation of the `fire` Method of the Petri Nets DSL Operations Semantics.**

the method `fire()` fires a given transition—i. e. moves the tokens from an input to an output place.

Figure 7 shows the implementation of the *fire* operation. First, the method checks if it is possible to *fire* the transition. If it is not, `false` is returned. Otherwise, `rc_transition` is borrowed to iterate in its `input_edges` attribute. For each *edge*, a number of tokens corresponding to the weight of the *edge* are removed from the mutably borrowed place. The same operation is performed on the places at the end of the `output_edges`, but instead of being removed, the tokens are added.

```typescript
import init, { PetrinetProxy } from "petrinet";

enum Orientation {
    toPlace = "toPlace",
    toTransition = "toTransition",
}

await init();
const petrinet = new PetrinetProxy();

petrinet.add_place("P1", 1);
petrinet.add_transition("T1");
petrinet.add_edge("P1", "T1", Orientation.toTransition, 1);
petrinet.fire("T1");
```

**Figure 8: Example of TypeScript Code Using the web Runtime Generated from the Petri Nets DSL Rust Implementation.**

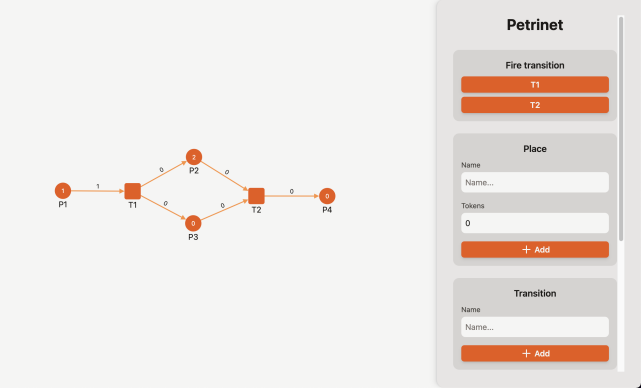


**Figure 9: Screenshot of a Graphical End-user web Tool Implemented Using the web Runtime Generated from the Petri Nets DSL Rust Implementation.**

At this point, we have a complete Rust implementation for the Petri nets DSL, which can be given to different toolchains to obtain different runtimes. For example, a web runtime can be generated using the wasm-pack toolchain, thus making it possible to integrate the DSL inside a web application. Such a generated web runtime includes both WebAssembly binaries and TypeScript glue code allowing easy use of the WebAssembly part. Figure 8 shows a simple example of using this web runtime in a TypeScript program. The module is imported, then initialized and finally, the WebAssembly functions exposed can be called from TypeScript. Pushing this idea further, Figure 9 shows a screenshot of a complete web graphical end-user tool[19] implemented in this fashion, which can both create and execute Petri nets models.

A native runtime can also be generated for the DSL using the Rust compiler, which can be directly imported into a Rust program to provide a native tool. Figure 10a and Figure 10b show an example of textual command-line end-user tool implemented using this runtime, which is also able to create and execute a Petri net model.

## 4 EVALUATION

In this section we present an initial evaluation of our approach through the following three research questions:

- *RQ 1:* Is the approach generic, i. e. can it be used to create different sorts of DSLs in Rust?
- *RQ 2:* How much manual development effort is avoided using the automated part of the approach (Ecore2Rust)?
- *RQ 3:* Can the approach be used to obtain different sorts of runtimes for a DSL, and can these runtimes be used to build tools for different environments?

### 4.1 Evaluation protocol

*Considered DSLs.* For this initial evaluation, we considered two common DSLs: a Petri nets DSL and a Finite State Machines (FSM) DSL. The considered Petri nets DSL was already presented and used as a running example in the previous section. The considered

[19] Code available under the Apache Licence 2.0: https://gitlab.univ-nantes.fr/E187954Y/petrinet-player.

```
Welcome to Petrinet CLI.

Use the following commands to build your Petri net:
  [1] add a place       [2] add a transition    [3] add an edge
  [4] fire a transition [5] display petrinet    [6] exit
1

Enter information about the place: <name> <tokens>
P1 2
PlaceAdded

Use the following commands to build your Petri net:
  [1] add a place       [2] add a transition    [3] add an edge
  [4] fire a transition [5] display petrinet    [6] exit
1

Enter information about the place: <name> <tokens>
P2 0
PlaceAdded

Use the following commands to build your Petri net:
  [1] add a place       [2] add a transition    [3] add an edge
  [4] fire a transition [5] display petrinet    [6] exit
2

Enter information about the transition: <name>
T1
TransitionAdded

Use the following commands to build your Petri net:
  [1] add a place       [2] add a transition    [3] add an edge
  [4] fire a transition [5] display petrinet    [6] exit
3

Enter information about the edge: <place_name> <transition_name> <orientation> <weight>
  Orientation is a boolean: 'true' for an edge pointing to a place and 'false' for an edge pointing to a transition.
P1 T1 false 1
EdgeAdded

Use the following commands to build your Petri net:
  [1] add a place       [2] add a transition    [3] add an edge
  [4] fire a transition [5] display petrinet    [6] exit
3

Enter information about the edge: <place_name> <transition_name> <orientation> <weight>
  Orientation is a boolean: 'true' for an edge pointing to a place and 'false' for an edge pointing to a transition.
P2 T1 true 1
EdgeAdded
```

**(a) Creation of a Petri net Model.**

```
Use the following commands to build your Petri net:
  [1] add a place       [2] add a transition    [3] add an edge
  [4] fire a transition [5] display petrinet    [6] exit
5
"P1[2] -(1)-> T1 ; T1 -(1)-> P2[0] ; "

Use the following commands to build your Petri net:
  [1] add a place       [2] add a transition    [3] add an edge
  [4] fire a transition [5] display petrinet    [6] exit
4

Enter information about the transition to fire: <name>
T1
FireSuccessful

Use the following commands to build your Petri net:
  [1] add a place       [2] add a transition    [3] add an edge
  [4] fire a transition [5] display petrinet    [6] exit
5
"P1[1] -(1)-> T1 ; T1 -(1)-> P2[1] ; "

Use the following commands to build your Petri net:
  [1] add a place       [2] add a transition    [3] add an edge
  [4] fire a transition [5] display petrinet    [6] exit
6

Exit...
```

**(b) Firing a Transition of the Created Model.**

**Figure 10: Screenshots of a Textual Command-line End-user Native Tool Implemented Using the Native Runtime Generated from the Petri Nets DSL Rust Implementation.**

FSM DSL is centered on the concepts of *State* and *Transition*, with a semantics describing how the current state of the FSM changes based on the executed transitions.

*Considered execution environments and target tool.* We considered the same two execution environments that have been used as examples in previous sections: a *native* environment, where the DSL runtime must be a native library (e. g. x86 instructions), and a *web* environment, where the DSL must be a library executable in a browser (e. g. JS and WebAssembly). For each environment, we aim to implement an end-user tool able both to create a model and execute it.

*Considered metrics.* To estimate development effort, we relied on measuring the number of lines of code using the *cloc* tool with default options, i. e. not counting empty lines and comments.

*Evaluation process.* We followed the following process for each considered DSL:

(1) We used the Ecore language to create an abstract syntax in the form of a metamodel.
(2) We applied *Ecore2Rust* on the Ecore abstract syntax to generate a Rust version of the abstract syntax.
(3) Using the abstract syntax generated code, we manually implemented an operational semantics fully in Rust, thus yielding a complete Rust implementation of the DSL.
(4) We measured both the amount of code generated and the amount of code manually written.
(5) Finally, for each considered execution environment:
   (a) We used the Rust compiler for generating a DSL runtime if considering a native environment, or wasm-pack if considering a web environment.
   (b) Using the resulting runtime, we manually implemented the considered end-user tool.

## 4.2 Results

In the following, we present the obtained results and discuss how they answer the considered research questions.

*RQ 1:* The approach was successfully applied to obtain the abstract syntax, the editing operations and the operational semantics in Rust of both considered DSLs. The resulting Rust implementations are publicly available on a GitLab instance[20][21].

*RQ 2:* Table 4 shows the amount of Rust code obtained for each DSL. We count separately the lines that were automatically generated (i. e. the abstract syntax) and the lines that were manually written (i. e. operational semantics and editing operations). We note that the approach generates between 23.68 % and 26.98 % of the total amount of code to get a complete DSL using Rust. Although

[20] https://gitlab.univ-nantes.fr/E187954Y/fsm-player
[21] https://gitlab.univ-nantes.fr/E187954Y/petrinet-player

| | FSM | Petri nets |
|---|---|---|
| Generated rows | 17 | 27 |
| Manually written rows | 43 | 87 |
| Percentage of generated code | 26.98 % | 23.68 % |

**Table 4: Proportion of Automatically Generated Code**

the approach requires the developer to write a significant part of the code, it should be noted that the remaining part requires less design work because it relies on automatically generated data structures.

*RQ 3:* Each DSL was successfully compiled in two forms: a native runtime and a web runtime. We built a variant of the considered tool for each runtime of each DSL: a textual command-line tool for the native environment and a graphical tool for the web environment. The tools obtained for the Petri nets DSL were already presented with Figures 9, 10a, and 10b. Figure 11 shows a screenshot of the graphical web end-user tool made for the FSM DSL using the generated web runtime, and Figures 12a and 12b screenshots of the textual native end-user tool made using the generated native runtime. All four end-user tools are available on a Gitlab instance[22][23].

Also note that for each DSL, resulting end-user tools rely on the same Rust implementation of the DSL, thus avoiding any error-prone re-implementation of the DSL for each new environment.

## 5 RELATED WORK

The Eclipse Modeling Framework (EMF), provides the official code generator for Ecore, targeting Java [12]. A few other approaches propose translations and implementations of Ecore in different programming languages. The .NET Modeling Framework (NMF) is able to generate C# code from Ecore [6]. The CrossEcore cross-platform modeling framework is also able to generate C#, but also TypeScript, JavaScript, and Swift from Ecore models with embedded OCL [11]. The PyEcore[24] framework provides an implementation of EMF in Python, making it possible to manipulate Ecore metamodels and models in Python. However, to our knowledge, we are the first approach targeting Rust, which focuses on generating different runtimes from the same DSL implementation.

We can also cite the work of Besnard et al. [1], where a native runtime and tool for the UML language was developped in C , thus making it able to execute UML models in a bare-metal embedded environment. However the code was written manually, and cannot be automatically generated from a DSL definition as with our approach.

## 6 CONCLUSION

We presented an approach to engineer a DSL using Rust, up to the generation of different DSL runtimes using the existing Rust toolchains. The approach includes a tool, Ecore2Rust, able to automatically translate an abstract syntax described in Ecore into Rust. To evaluate our solution, we applied it to two DSLs: FSM and Petri nets. Results show that the approach works for both DSLs, reduces the amount of code to write manually, and can indeed be used to create tools for different execution environments (native and web).

To pursue this work, different research directions are possible. The evaluation should include a greater diversity of DSLs, including more complex cases such as parts of the Unified Modeling Language (UML). The Ecore2Rust tool can be extended to support more parts of the Ecore language. It would also be interesting to automatically generate (part of) the Rust operational semantics, for instance using a metalanguage such as ALE [9] as source language.

[22] https://gitlab.univ-nantes.fr/E187954Y/petrinet-player
[23] https://gitlab.univ-nantes.fr/E187954Y/fsm-player
[24] https://github.com/pyecore/pyecore

## REFERENCES


[1] Valentin Besnard, Matthias Brun, Frédéric Jouault, Ciprian Teodorov, and Philippe Dhaussy. 2018. Unified LTL Verification and Embedded Execution of UML Models. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems* (Copenhagen, Denmark) *(MODELS '18)*. Association for Computing Machinery, New York, NY, USA, 112–122. https://doi.org/10.1145/3239372.3239395

[2] Erwan Bousse, Thomas Degueule, Didier Vojtisek, Tanja Mayerhofer, Julien Deantoni, and Benoit Combemale. 2016. Execution framework of the GEMOC studio (tool demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. ACM. https://doi.org/10.1145/2997364.2997384

[3] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. 2015. Evaluating and comparing language workbenches. *Computer Languages, Systems and Structures* 44 (Dec. 2015), 24–47. https://doi.org/10.1016/j.cl.2015.08.007

[4] Martin Fowler. 2010. *Domain-specific languages.* Pearson Education.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software.* AddisonWesley Professional.

[6] Georg Hinkel. 2018. NMF: A multi-platform Modeling Framework. In *Theory and Practice of Model Transformations: 11th International Conference, ICMT 2018, Held as Part of STAF 2018, Toulouse, France, June 25-29, 2018. Proceedings*, Arend Rensink and Jesus Sanchez Cuadrado (Eds.). Springer International Publishing.

[7] Steven Kelly, Kalle Lyytinen, and Matti Rossi. 1996. MetaEdit+ A fully configurable multi-user and multi-tool CASE and CAME environment. In *Notes on Numerical Fluid Mechanics and Multidisciplinary Design*. Springer International Publishing, 1–21. https://doi.org/10.1007/3-540-61292-0_1

[8] Paul Klint, Tijs van der Storm, and Jurgen Vinju. 2009. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE. https://doi.org/10.1109/scam.2009.28

[9] Manuel Leduc, Thomas Degueule, Benoit Combemale, Tijs van der Storm, and Olivier Barais. 2017. Revisiting Visitors for Modular Extension of Executable DSMLs. In *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. IEEE. https://doi.org/10.1109/models.2017.23

[10] Marjan Mernik, Jan Heering, and Anthony M Sloane. 2005. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)* 37, 4 (2005), 316–344.

[11] Simon Schwichtenberg, Ivan Jovanovikj, Christian Gerth, and Gregor Engels. 2018. CrossEcore: An Extendible Framework to Use Ecore and OCL across Platforms. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 292–293. https://doi.org/10.1145/3183440.3194976

[12] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. 2008. *EMF: eclipse modeling framework.* Pearson Education.

[13] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart CL Kats, Eelco Visser, and GH Wachsmuth. 2013. DSL engineering-designing, implementing and using domain-specific languages. (2013).

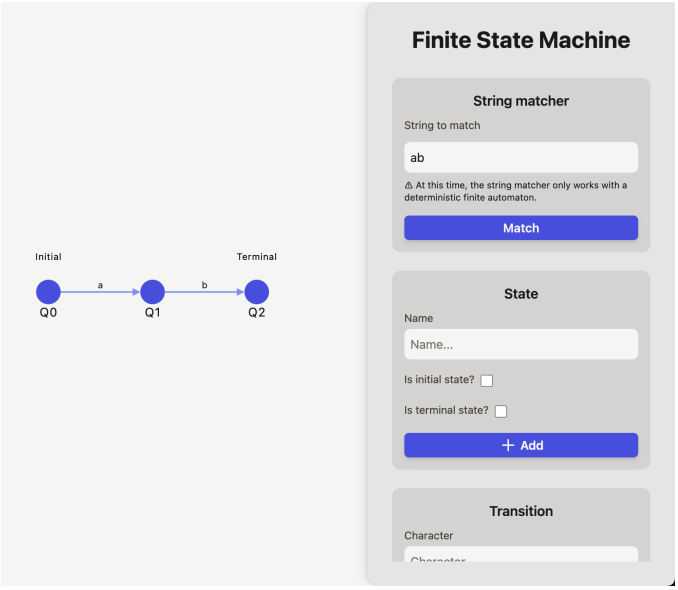


**Figure 11: Screenshot of a Graphical End-user web Tool Implemented Using the web Runtime Generated from the FSM DSL Rust Implementation.**

```
Welcome to Finite State Machine CLI.

Use the following commands to build your Finite State Machine:
  [1] add a node        [2] add a transition    [3] string matcher
  [4] display petrinet  [5] exit
1

Enter information about the node: <name> <is_initial> <is_final>
Q0 true false
NodeAdded

Use the following commands to build your Finite State Machine:
  [1] add a node        [2] add a transition    [3] string matcher
  [4] display petrinet  [5] exit
1

Enter information about the node: <name> <is_initial> <is_final>
Q1 false false
NodeAdded

Use the following commands to build your Finite State Machine:
  [1] add a node        [2] add a transition    [3] string matcher
  [4] display petrinet  [5] exit
1

Enter information about the node: <name> <is_initial> <is_final>
Q2 false true
NodeAdded

Use the following commands to build your Finite State Machine:
  [1] add a node        [2] add a transition    [3] string matcher
  [4] display petrinet  [5] exit
2

Enter information about the transition: <char> <input_node> <output_node>
a Q0 Q1
TransitionAdded

Use the following commands to build your Finite State Machine:
  [1] add a node        [2] add a transition    [3] string matcher
  [4] display petrinet  [5] exit
2

Enter information about the transition: <char> <input_node> <output_node>
b Q1 Q2
TransitionAdded
```

**(a) Creation of an FSM Model.**

```
Use the following commands to build your Finite State Machine:
  [1] add a node        [2] add a transition    [3] string matcher
  [4] display petrinet  [5] exit
4
(Q0) -a-> (Q1)
(Q1) -b-> (Q2)


Use the following commands to build your Finite State Machine:
  [1] add a node        [2] add a transition    [3] string matcher
  [4] display petrinet  [5] exit
3

Enter a word to test if the FSM recognizes it: <word>
  Note that string matcher only works on non-deterministic finite automata
ab
ab
(Q0) -a-> (Q1)
(Q1) -b-> (Q2)

StringMatchSuccessful

Use the following commands to build your Finite State Machine:
  [1] add a node        [2] add a transition    [3] string matcher
  [4] display petrinet  [5] exit
3

Enter a word to test if the FSM recognizes it: <word>
  Note that string matcher only works on non-deterministic finite automata
ac
ac
(Q0) -a-> (Q1)
(Q1) -b-> (Q2)

StringMatchFailed

Use the following commands to build your Finite State Machine:
  [1] add a node        [2] add a transition    [3] string matcher
  [4] display petrinet  [5] exit
5

Exit...
```

**(b) String Matching on the Created Automaton.**

**Figure 12: Screenshots of a Textual Command-line End-user Native Tool Implemented Using the Native Runtime Generated from the FSM DSL Rust Implementation.**