Solving the TTC'14 FIXML Case Study with SIGMA

Filip Křikava
University Lille 1 - LIFL, France
INRIA Lille, Nord Europe
filip.krikava@inria.fr

Philippe Collet
Université Nice - Sophia Antipolis, France
CNRS, I3S, UMR 7271
philippe.collet@unice.fr

In this paper we describe a solution for the *Transformation Tool Contest 2014* (TTC'14) FIXML case study using SIGMA, a family of Scala internal *Domain-Specific Languages* (DSLs) that provides an expressive and efficient API for model consistency checking and model transformations. We solve both the core transformation task and its three extensions.

1 Introduction

In this paper we describe our solution for the TTC'14 FIXML case study [3] using the SIGMA internal DSLs [2]. In addition to solving the core tasks that consists of generating Java, C# and C++ code from FIXML messages structure, we also solve all the proposed extensions for determining appropriate types of element attributes, generating C code and generic FIXML schema transformation. Our solution supports array generation for multiple sibling XML nodes that share the same name, and most notably it generates proper constructor calls, keeping the initial values as they occur in the XML document. Purposely, we choose a non-trivial object-oriented model in order to demonstrate that SIGMA can be easily applied in complex transformations. Finally, for all target languages, the generated code is complete, including all necessary statements so it compiles without any problems or warning[1].

The solution was developed in SIGMA, a family of Scala[2] internal DSLs for model manipulation tasks such as model validation, model to model (M2M), and model to text (M2T) transformations. Scala is a statically typed production-ready *General-Purpose Language* (GPL) that supports both object-oriented and functional styles of programming. It uses type inference to combine static type safety with a *"look and feel"* close to dynamically typed languages. SIGMA DSLs are embedded in Scala as a library allowing one to manipulate models using high-level constructs similar to the ones found in the external model manipulation DSLs such as ETL or ATL [3]. The intent is to provide an approach that developers can use to implement many of the practical model manipulations within a familiar environment, with a reduced learning overhead as well as improved usability and performance. The solution is based on the *Eclipse Modeling Framework* (EMF), which is a popular meta-modeling framework widely used in both academia and industry, and which is directly supported by SIGMA.

The complete source code is available on Github as well as directly runnable from a SHARE environment[4].

[1]Tested using OpenJDK 1.7, Mono 3.2 and Clang 6.0 on OSX 10.9
[2]http://scala-lang.org
[3]Epsilon – https://www.eclipse.org/epsilon/, ATL – https://www.eclipse.org/atl/
[4]Github – http://bit.ly/1mHSCHY, SHARE – http://bit.ly/1sSWDeB

Submitted to:
TTC 2014

2 Solution Description

The core problem of this case study is generating source code from a FIXML message structure. The input is a file representing an FIXML 4.4 message [1] and the output is some Java, C# and C++ sources that represent the structure of the given FIXML message. As suggested, our solution is realized by a systematic model transformation from an XML file to an XML model, which is transformed into an object-oriented language model, from which we serialize into source code.

Prerequisites In SIGMA, EMF models are aligned with Scala through automatically generated extension traits (placed in an src-gen folder) that allows for a seamless model navigation and modification using standard Scala expressions. This includes omitting `get` and `set` prefixes, convenient first-order logic collection operations (*e.g.*, `map`, `filter`, `reduce`), and first-class constructs for creating new model elements.

2.1 FIXML XML Message to XML Model

The parsing of an XML document is handled by a Scala library. Therefore this tasks in essentially a M2M transformation between the Scala XML model and the XML model specified in the case study description. This is a trivial operational-style transformation that has been realized by the `FIXMLParser` class.

2.2 XML Model to ObjLang Model

The ObjLang meta-model chosen for this solution originates from the Featherweight Java model [5]. It provides a reasonable abstraction for an object-oriented programming language, supporting basic classes, fields and expressions.

In SIGMA, a M2M transformation is represented as a Scala class that inherits from the `M2MT` base class. Concretely, the `XMLMM2ObjLang` our transformation class is defined as:

```
class XMLMM2ObjLang extends M2MT with XMLMM with ObjLang { // mix-in generated extensions
  // rule definitions
}
```

Within the class body, an arbitrary number of transformation rules can be specified as methods using parameters to define the transformation source-target relation. For example, the first rule of the transformation from `XMLNode` into a `Class` is defined as:

```
1 def ruleXMLNode2Class(s: XMLNode, t: Class) {
2   s.allSameSiblings foreach (associate(_, t))
3   t.name = s.tag
4   t.members ++= s.sTargets[Constructor]
5   t.members ++= s.allAttributes.sTarget[Field]; t.members ++= s.allSubnodes.sTarget[Field]
6 }
```

This rule represents a matched rule which is automatically applied for all matching elements. When such a rule is executed, the transformation engine first creates all the defined target elements and then calls the method whose body populates their content using arbitrary Scala code. A matched rule is applied once and only once for each matching source element, creating a 1:1 or 1:N mapping. However, in the current scenario, all XML node siblings with the same tag name should be mapped into the same class (N:1 mapping). This can be done by explicitly associating the siblings to the same class (line 2)[6]. The `sTarget(s)` methods are used to relate the corresponding target element(s)

[5] `http://bit.ly/1mHRkwM`

[6] The `allSameSiblings` is a helper collecting all same named siblings.

that has been already or can be transformed from source element(s). On lines 4 and 5 the use of these methods will populate the content of the newly created class by in turn executing the corresponding rules, *i.e.* `ruleXMLNode2DefaultConstructor`, `ruleXMLNode2NoneDefaultConstructor`, `ruleXMLAttribute2Field` and `ruleXMLNode2Field`.

The last rule converting XML nodes into fields has to handle multiple same-tag siblings. While the case description proposes to use either multiple fields or a collection, the former brings a scalability problem since in Java, there is a limit of the maximum number of method parameters already exceeded by the test case 5. Therefore we have opted for the latter and use arrays. Moreover, even though it has not been specifically requested in the case study, our transformation keeps the default values of the attributes of the different nodes and use them for constructing the instances. This is done by the `ruleXMLNode2ConstructorCall` rule.

2.3 ObjLang Model to Source code

This task involves transforming the ObjLang model into source code. SIGMA provides a template-based code-explicit[7] M2T transformation DSL that relies on Scala support for multi-line string literal and string interpolation. Since we target multiple programming languages, we organize the code generation in a set of Scala classes and use inheritance and class mix-ins to modularly compose configurations for the respective languages. In the base classes we define methods that synthesize expressions and data types (**`BaseObjLangMTT`**) and abstract a class generation (**`BaseObjLang2Class`**). Then we use these bases to configure concrete transformations.

The class **`ObjLang2Java`** contains the Java language specifics and it is almost the same as the C# **`ObjLang2CSharp`** generator. For C++, the situation is more complicated, since next to the class implementation (**`ObjLang2CPPClassImpl`**) a header file has to be generated (**`ObjLang2CPPClassHeader`**). Furthermore, the ObjLang model is less suitable for C++ classes and thus extra work has to be performed in the M2T transformation. The following is an example of a C++ header generator:

```scala
class ObjLang2CPPClassHeader extends BaseObjLang2Class with ObjLang2CPP with ObjLang2CPPHeader {
  override def header = {
    super.header
    source.fields map (_.type_) collect {
      case x: Class => x
    } map (_.cppHeaderFile) foreach { hdr =>
      !s"#include ${hdr.quoted}"
    }
    !endl
  }

  override def genFields = {
    !"public:" indent {
      super.genFields
    }
  }

  override def genConstructors = {
    !"public:" indent {
      super.genConstructors
    }
  }
}
```

It mixes in basic traits and then overrides the template that generates the different segments of the source code. For example, in C++ we need to add the `public` keyword before the fields and constructor declarations. The unary `!` (bang) operator provides a convenient way to output text. The `s` string prefix denotes an interpolated string, which can include Scala expressions.

[7]It is the output text instead of the transformation code that is escaped.

3 Extensions

In this section we describe our solutions to the case study extensions. Each extension has been implemented as a separate project on GitHub and therefore an interested reader can easily see what exactly has been changed.

3.1 Extension 1 - Selection of Appropriate Data Types

In this extension we use a simple heuristics to find an appropriate type for a field based on observed attribute values. We only cover numbers (`int`, `long`, `double`), but since the process is mechanical, it can be easily extended to cover all XML Schema data types. The ObjLang meta-model was extended with new expressions representing the new type literals. In the M2M transformation we added a function, `guessDataType`, which uses regular expressions to guess a data type based on a single attribute value. An attribute can occur multiple times with different values and therefore we need to consider all values at the same time in order to infer a type that is wide enough. An overridden function `guessDataType` takes a sequence of attribute values, guesses their individual types and then simply reduces the type to the largest one.

3.2 Extension 2 - Extension to Additional Languages

This extension adds a support for the C language. Since C is not an object-oriented language, more work has to be done in the code generator part, but the M2M transformations remain untouched. Instead of classes, we generate C structs with appropriate functions simulating object constructors. In order to simplify the code generator, we use a helper function that allows us to initialize arrays using simple expressions, which is not directly supported by the C language or by its standard library. Despite the lack of object orientation, our organization of the M2T transformation templates makes the implementation only 36 lines longer than the C++ version.

3.3 Extension 3 - Generic Transformation

Essentially, a generic FIXML Schema to ObjLang model transformation means creating a generic XML schema to ObjLang transformation. Such a task is far from being trivial and it would require a significant engineering effort. Therefore we have chosen an alternative solution in which we transform Java classes generated from an XML schema by the JAXB tool[8]. The advantage of this solution is that the JAXB already does all the hard work of parsing XML schema, resolving the element inheritance, substitutability, data types and others. The resulting Java classes in fact represents an object-oriented model and therefore the actual M2M transformation into ObjLang is rather straight forward. Finally, the JAXB can be thought of as a another model transformation. Therefore our solution is still within the model-driven engineering domain while demonstrating a strong advantage of internal DSL in reusing very easily another API with the host language.

The new input is a location of the FIXML XSD files and the new transformation workflow consists of (1) XSD to Java sources using JAXB, (2) Java source to Java classes using a Java compiler, (3) Java classes to ObjLang, (4) ObjLang to source source. The ObjModel had to be extended to cover enumerated types a notion of inheritance and abstract classes. The new transformation (**`Java2ObjLang`**) is about 30% smaller than the original transformation and arguably less complex. It also demonstrates SIGMA support for manipulating different models than EMF, *i.e.*, Java model

[8]Java Architecture for XML Binding `https://jaxb.java.net/`

implemented using Java reflection (`JavaClassModel`). The M2T transformation remained mostly untouched apart from the model extensions. It is important to note that the C code generator supports neither class inheritance nor abstract classes.

4 Evaluation and Conclusion

We evaluate our solution to the core problem using the evaluation criteria proposed in the case study description [3].

- The *complexity* as the number of operator occurrences, features and entity type name references in the specification expressions. To the best of our knowledge there is no tool providing this metric for Scala code. We therefore only provide our own estimate for the M2M transformation, which contains about 450 expressions and uses 18 meta-models classes with 23 references.
- The *accuracy* measures the syntactical correctness of the generated source code and how well the code represents the FIXML messages. The generated code compiles for all languages without any warning nor any special compiler settings. Using arrays to represent same-tag sibling nodes improves the quality and scalability of the code which is further enhanced by data type heuristics for field types. Finally, we have also implemented the generic FIXML Schema transformation that should result in a complete representation of FIXML messages in the different languages.
- The development effort is estimated to be about 15 person-hours for the core problem.
- The *fault tolerance* is high since the Scala XML library can detect invalid XML with accurate parsing errors.
- For all test cases (1, 2, 5 and 6), the *execution time* is about 7500ms for all the transformations on SHARE.
- *Modularity* for the M2M transformation is $1 - \frac{d}{r} = \frac{7}{8} = 0.125$, where d is the number of dependencies between rules and r is the number of rules.
- The level of *abstraction* for both the M2M and M2T transformations is medium since the rules are defined declaratively (high abstraction), but their content is an imperative code (medium).

Despite that we opted for a complex ObjLang model, the resulting transformations are rather expressive and quite concise. The complete implementation of the core problem consists of 500 lines of Scala code[9]. This FIXML case study provides a good illustration for some of the capabilities of an internal DSL approach to model manipulations in the model-driven engineering domain.

Acknowledgments This work is partially supported by the Datalyse project (`www.datalyse.fr`).

References

[1] FIXML (2004): *FIXML 4.4 Schema Version Guide*.

[2] Filip Krikava, Philippe Collet & Robert B France (2014): *Manipulating Models Using Internal Domain-Specific Languages*. In: *Symposium on Applied Computing (SAC), track on Programming Languages (PL)*, SAC.

[3] K. Lano, S. Yassipour-Tehrani & K. Maroukian (2014): *Case study: FIXML to Java, C# and C++*. In: *Transformation Tool Contest 2014*.

[9]The extension 1 consists of 550, extension 2 of 720 and extension 3 of 770 source lines of code.

A Meta-Models

A.1 XML Meta-Model

The XML model specified in the case study description [3].

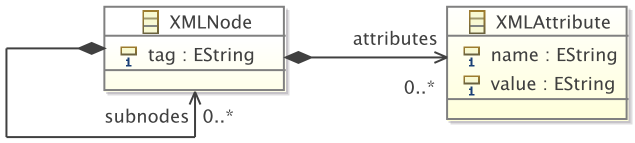

Figure 1: XML meta-model

A.2 ObjLang Meta-Model

The meta-model representing an object oriented language originating from the Featherweight Java model, concretely from the version available at the EMFtext website[10].

B XML File to XML Model Transformation

```
protected def parseNodes(nodes: Iterable[Node]): Iterable[XMLNode] = {
  val elems = nodes collect { case e: Elem => e }

  for (elem <- elems) yield XMLNode(
    tag = elem.label,
    subnodes = parseNodes(elem.child),
    attributes = parseAttributes(elem.attributes))
}

protected def parseAttributes(metaData: MetaData) =
  metaData collect {
    case e: xml.Attribute => XMLAttribute(name = e.key, value = e.value.toString)
  }
```

C XML Model to ObjLang Model Transformation Rules

```
def ruleXMLNode2DefaultConstructor(s: XMLNode, t: Constructor) {
  s.allSameSiblings foreach (associate(_, t))
}
```

```
def ruleXMLNode2NonDefaultConstructor(s: XMLNode, t: Constructor) = guardedBy {
  !s.isEmptyLeaf
} transform {

  s.allSameSiblings foreach (associate(_, t))
```

[10] http://bit.ly/1mHRkwM

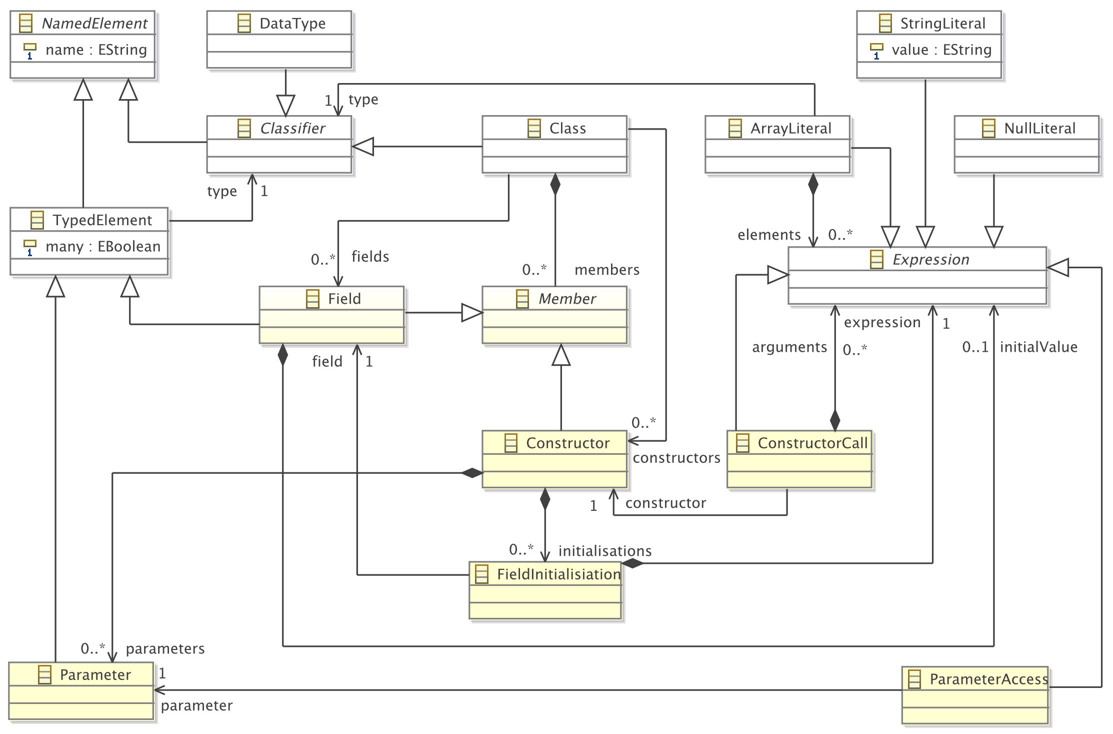

Figure 2: ObjLang meta-model

```

  for (e <- (s.allAttributes ++ s.allSubnodes.distinctBy(_.tag))) {
    val param = e.sTarget[Parameter]
    val field = e.sTarget[Field]

    t.parameters += param
    t.initialisations += FieldInitialisiation(
      field = field,
      expression = ParameterAccess(parameter = param))
  }
}
```

```
def ruleXMLAttribute2ConstructorParameter(s: XMLAttribute, t: Parameter) {
  t.name = checkName(s.name)
  t.type_ = s.sTarget[Field].type_
}
```

```
def ruleXMLNode2ConstructorParameter(s: XMLNode, t: Parameter) {
  val field = s.sTarget[Field]

  t.name = field.name
  t.many = field.many
  t.type_ = field.type_
}
```

```
@LazyUnique
def ruleXMLAttribute2Field(s: XMLAttribute, t: Field) {
  t.name = checkName(s.name)

  t.type_ = DTString
  t.initialValue = StringLiteral(s.value)
}
```

```
@LazyUnique
def ruleXMLNode2Field(s: XMLNode, t: Field) {
  val allSiblings = s.allSameSiblings
  allSiblings foreach (associate(_, t))

  t.type_ = s.sTarget[Class]

  val groups = (s +: allSiblings) groupBy (_.eContainer)
  val max = groups.values map (_.size) max

  if (max > 1) {
    t.name = s.tag + "_objects"
    t.many = true
    val init = ArrayLiteral(type_ = s.sTarget[Class])
    val siblings = groups(s.eContainer)

    init.elements ++= siblings.sTarget[ConstructorCall]
    init.elements ++= 0 until (max - siblings.size) map (_ => NullLiteral())
    t.initialValue = init
  } else {
    t.name = s.tag + "_object"
    t.initialValue = s.sTarget[ConstructorCall]
  }
}
```

```
@Lazy
def ruleXMLNode2ConstructorCall(s: XMLNode, t: ConstructorCall) {
  val constructor = s.sTargets[Constructor]
    .find { c =>
      (c.parameters.isEmpty && s.isEmptyLeaf) ||
      (c.parameters.nonEmpty && !s.isEmptyLeaf)
    }
    .get

  t.constructor = constructor

  t.arguments ++= {
    for {
      param <- constructor.parameters
      source = param.sSource.get
    } yield {
      source match {
        case attr: XMLAttribute =>
          // we can cast since attributes have always primitive types
          val dataType = param.type_.asInstanceOf[DataType]

          s.attributes
            .find(_.name == attr.name)
            .map { local => StringLiteral(local.value) }
            .getOrElse(NullLiteral())

        case node: XMLNode =>
          s.subnodes.filter(_.tag == node.tag) match {

            case Seq() if !param.many =>
              NullLiteral()
            case Seq(x) if !param.many =>
              x.sTarget[ConstructorCall]
            case Seq(xs @ _*) =>
              val groups = (node +: node.allSameSiblings) groupBy (_.eContainer)
            val max = groups.values map (_.size) max

            val init = ArrayLiteral(type_ = param.type_)
            init.elements ++= xs.sTarget[ConstructorCall]
            init.elements ++= 0 until (max - xs.size) map (_ => NullLiteral())
            init
          }
        }
      }
    }
}
```

D Handling Constructor Arguments

The number of same-tag sibling nodes can vary within a parent node. For example:

```
<Pty ID="OCC" R="21"/>
<Pty ID="C" R="38">
         <Sub ID="ZZZ" Typ="2"/>
</Pty>
<Pty ID="C" R="38" Z="Q">
  <Sub ID="ZZZ" Typ="2"/>
  <Sub ID="ZZZ" Typ="3" Oed="X"/>
</Pty>
```

The `Sub` should be represented by an array field and the default initialization of `PosRpt` should equal to the following (in Java):

```
public Pty[] Pty_objects = new Pty[] {
  new Pty("OCC", "21", null, new Sub[] { null, null }),
  new Pty("C", "38", null, new Sub[] { new Sub("ZZZ", "2", null), null }),
  new Pty("C", "38", "Q", new Sub[] { new Sub("ZZZ", "2", null), new Sub("ZZZ", "3", "X") })
};
```

Note that the first and second instances of `Pty` contain two and one `null` respectively in the place of missing `Sub` subnode.

The `ConstructorCall` used for field initializations in the `ruleXMLNode2Field` is created from an XML node using the last rule in the transformation:

```
@Lazy
def ruleXMLNode2ConstructorCall(s: XMLNode, t: ConstructorCall) {
  val constructor = s.sTargets[Constructor]
    .find { c =>
      (c.parameters.isEmpty && s.isEmptyLeaf) ||
      (c.parameters.nonEmpty && !s.isEmptyLeaf)
    }
    .get

  t.constructor = constructor

  t.arguments ++= {
    for {
      param <- constructor.parameters
      source = param.sSource.get
    } yield {
      source match {
        case attr: XMLAttribute =>
          // we can cast since attributes have always primitive types
          val dataType = param.type_.asInstanceOf[DataType]

          s.attributes
            .find(_.name == attr.name)
            .map { local => StringLiteral(local.value) }
            .getOrElse(NullLiteral())

        case node: XMLNode =>
          s.subnodes.filter(_.tag == node.tag) match {

            case Seq() if !param.many =>
              NullLiteral()
            case Seq(x) if !param.many =>
              x.sTarget[ConstructorCall]
            case Seq(xs @ _*) =>
              val groups = (node +: node.allSameSiblings) groupBy (_.eContainer)
              val max = groups.values map (_.size) max

              val init = ArrayLiteral(type_ = param.type_)
              init.elements ++= xs.sTarget[ConstructorCall]
              init.elements ++= 0 until (max - xs.size) map (_ => NullLiteral())
              init
          }
      }
    }
  }
}
```

First we need to find which constructor shall be used depending on whether the given XML node (or any of its same-tag siblings) contains any attributes or subnodes. Next, we need to resolve the arguments for the case of non-default constructor. We do this by using the sources, *i.e.*, the source elements (XML node or XML attribute) that were used to create the constructor parameters. SIGMA provides `sSource` method that is the inverse of `sTarget` call with the difference that it will not trigger any rule execution. In the pattern matching we need to cover all possible cases such as an attribute defined locally or an attribute defined in a same-tag sibling, thus using `null` for its initialization.

E Data Type Heuristics

```
  // basic types
  val DTString = DataType(name = "string")
  val DTDouble = DataType(name = "double")
  val DTLong = DataType(name = "long")
  val DTInteger = DataType(name = "int")

  // it also stores the promotion ordering from right to left
  val Builtins = Seq(DTString, DTDouble, DTLong, DTInteger)

  private val PDouble = """([+-]?\d+.\d+)""".r
  private val PInteger = """([+-]?\d+)""".r

  def guessDataType(value: String): DataType = value match {
    case PDouble(_) => DTDouble
    case PInteger(_) => Try(Integer.parseInt(value)) map (_ => DTInteger) getOrElse (DTLong)
    case _ => DTString
  }

  def guessDataType(values: Seq[String]): DataType =
    values map guessDataType reduce { (a, b) =>
      if (Builtins.indexOf(a) < Builtins.indexOf(b)) a else b
    }
```

F Generating C Code

Input document:

```xml
<Pty ID="C" R="38">
  <Sub ID="ZZZZZ" Typ="2"/>
  <Sub ID="ZZZ" Typ="2"/>
</Pty>
```

Generated C code:

```c
#ifndef _Pty_H_
#define _Pty_H_

#include <stdlib.h>

#include "Sub.h"

typedef struct {
  char* _R;
  char* _ID;
  Sub** Sub_objects;
} Pty;

Pty* Pty_new();
Pty* Pty_init_custom(Pty* this, char* _R, char* _ID, Sub** Sub_objects);
Pty* Pty_init(Pty* this);

#endif // _Pty_H_
```

```c
#include "arrays.h"

#include "Pty.h"

Pty* Pty_new() {
  return (Pty*) malloc(sizeof(Pty));
}

Pty* Pty_init_custom(Pty* this, char* _R, char* _ID, Sub** Sub_objects) {
  this->_R = _R;
  this->_ID = _ID;
  this->Sub_objects = Sub_objects;
  return this;
}

Pty* Pty_init(Pty* this) {
  this->_R = "21";
  this->_ID = "OCC";
  this->Sub_objects = (Sub**) new_array(2, Sub_init_custom(Sub_new(), "2", "ZZZ"), NULL);
  return this;
}
```